Hieu Tran

Implementation of high integrity file system for serial flash memories

Subtitle

Metropolia University of Applied Sciences

Bachelor

Information Technology

Thesis

Date

|  |  |  |
| --- | --- | --- |
| Author(s)  Title | | Hieu Tran  Implementation of high integrity file system for serial flash memories |
| Number of Pages  Date | | xx pages + x appendices  5 May 2010 |
| Degree | | Bachelor |
| Degree Programme | | Information Technology |
| Specialisation option | | Smart Systems |
| Instructor(s) | | Keijo Länsikunnas, Title (for example: Project Manager)  First name Last name, Title (for example: Principal Lecturer) |
|  | | |
| Keywords |  | |

**Contents**

[1 Introduction 1](#_Toc24242987)

[2 Theoretical Background 2](#_Toc24242988)

[2.1 File System 2](#_Toc24242989)

[2.1.1 Definition 2](#_Toc24242990)

[2.1.2 File and Directory 3](#_Toc24242991)

[2.1.3 Terminologies 4](#_Toc24242992)

[2.1.4 Operations 5](#_Toc24242993)

[2.1.5 Existing File Systems 8](#_Toc24242994)

[2.2 Block-based File System 8](#_Toc24242995)

[2.2.1 Internal Components 9](#_Toc24242996)

[2.2.2 Advantages 10](#_Toc24242997)

[2.2.3 Disadvantages 11](#_Toc24242998)

[2.3 Flash Memory 11](#_Toc24242999)

[2.3.1 Operating Principle 13](#_Toc24243000)

[2.3.2 Hardware Limitation 16](#_Toc24243001)

[2.3.3 Managed Flash vs. Unmanaged Flash 18](#_Toc24243002)

[3 Design and Implementation 19](#_Toc24243003)

[3.1 Design Overview 19](#_Toc24243004)

[3.2 Node Data Structure 20](#_Toc24243005)

[3.3 Operations 23](#_Toc24243006)

[3.4 Garbage Collection 25](#_Toc24243007)

[4 Testing result 26](#_Toc24243008)

[4.1 Hardware in use 26](#_Toc24243009)

[4.2 Functionalities 27](#_Toc24243010)

[5 Discussion 27](#_Toc24243011)

[5.1 What design goal has been achieved 27](#_Toc24243012)

[5.2 Limitation of current design 27](#_Toc24243013)

[5.3 Further development 28](#_Toc24243014)

[5.4 What I achieve with this project 28](#_Toc24243015)

[6 Conclusion 28](#_Toc24243016)

Appendices

Appendix 1. Title of the Appendix

Appendix 2. Title of the Appendix

Maybe have

Introduction

File system Background

Flash Technology Background

# Introduction

Since the birth of modern computers based on abstract model of Turing machine in 1936 [1], computers have evolved significantly from the purpose of doing arithmetic computations into general purpose that could solve any mankind’s daily life problems; and the principle for this leap is the way computers treat input information, including computing, storing and manipulating data. Computers store data in a device called storage, which advances from punch cards, magnetic tapes to hard disks, semiconductor chips as today main storage device types.

In the early days, when computers only performed one computation, the entire storage operated like on big file; and data started at the beginning of storage, then filled up in order up to the storage capacity as output was produced. However, as computational power and storage capacity improved, it became possible, and useful to store more than one file at a time. Unfortunately, storage devices have no notion of files since they are only mechanisms for storing binary bits, hence the need for a system that can keep track of all files in the storage emerged, and it is known as file system (FS).

There are many types of computer storage were invented and each one of them was made from completely different materials with distinctive attributes. Thus, each type of storage device requires specific file systems that have the best knowledge of the underlying structure to get the most optimal functionalities and performance.

The most recent technology in storage device is flash memory, which was built on the ground of semiconductor material, and this kind of memory is quickly becoming the most popular computer storage. Much like transistor count and Moore’s law, memory and storage technologies have followed a similar exponential trend and as of 2017, NAND and NOR flash jointly accounted for 49.1 billion US dollars [2] in total 110 billion US dollars of Worldwide Memory Market [3].

<fig1> <ref3>

The motivation for this thesis topic started from current work environment of the author at Tuxera Inc. The author’s workplace is a software company located in Espoo, Finland who expertizes in embedded file systems and storage solutions [4]. In the path of gaining more knowledge to benefit daily work tasks, the author has set the objective to write his own file system with flash memory as the target storage. By following his objective, the author will learn about basics of file system design, which is the main product Tuxera Inc. offers, and understand technology behind the latest generation of storage device, flash memory.

The goal of this thesis topic is to program a simple but functional file system software for flash memory that could be used in low resource embedded applications. Two requirements this project design need to tackle are overcoming wear limitation of flash technology which will be discussed in Chapter ??? and preserving data integrity in any chance of power failure or disruptive communication.

# Theoretical Background

## File System

### Definition

File system is a piece of software that allows Operating System (OS) to manage data on permanent storages. These permanent storages, which often are optical disks, hard disks, flash storage drives, etc., are used to store computer data and information for long-term even without electrical power <ref5>. Therefore, the main function of file systems is providing a way to organize, store, retrieve, and manipulate data and information on such kind of storages. To bring the most natural experience for end users when storing and retrieving computer data to and from persistent storage devices, hierarchy concept has been adapted to the computer world with two main components: file and directory <ref6>. Thus, file systems are usually built around what is called file and directory.

### File and Directory

As introduced above in Chapter 2.1, file system is formed from the concept of computer file and directory. From human world, books in library are stored with careful thought about ease for later look up and retrieval, hence they are arranged into shelves sorted by family name of the authors or categorized by their subjects. File system can be compared to a library where a file is a book that has individual piece of information and directory is a shelf that holds multiple books or files that share common attributes. This relationship between file and directory makes organizing and looking for computer data more modular.

<fig2>

From file system point of view, a file has information used by the computer, and in order to query for the exact piece of data stored in a file, a filename must be assigned to the file. As a result, the job of file systems is storing files, which includes the raw data or "stream of bytes" and its names, on the storage device, then should be able to return back the content of the file when computer requests by passing the filename.

Most computer users are familiar with functionality of a file or a directory from direct experience when using the computers. Computer programs write data permanently to storage drive into files, which can be simply seen as a piece of information. This piece can be some texts, graphical images, data structures, any combination of data types or arbitrary bytes only the program can interpret.

Dominic Giampaolo hinted the important role of file system when storing file on physical medium: "A file appears as a continuous stream of bytes at higher levels, but the blocks that contain the file data may not be contiguous on disk." <ref7 p.11>. File systems is responsible for mapping file’s logical memory to physical memory on disk, thus when there is a write request, correct physical blocks will be modified and data on corresponding blocks needs to be returned in case of read request.

Directory, similar to the term Folder in Windows Operating System, is the way file system organize multiple files. Directory can be considered a file, but its content is a list of names, these names can be name of files or names of sub-directories (directories living under a directory). The policy of storing this name list in storage plays a crucial part in file lookup performance. For instance, if the list is unsorted, searching for a file name in the list must scan through the whole list and in the worst-case scenario the wanted file appears to be the file in the list. Therefore, normal file systems take advantage of sorted data structure when storing content of directory for efficiency in lookup operation.

### Terminologies

It is necessary to define some computer terms referring to certain concepts that make up a file system for better understanding the relationship between storage devices and file systems in computers. The list below is put down with dependency ordering, previous definitions are building blocks for the next ones.

(INTEGRITY libguide p.96 file system terms: file system implementation, file system instance, metadata, mount, partition, root file system)

* Partitions: are divisions or parts of a real hard disk drive. A partition is a logical separation from the whole drive, but it appears as though the division creates multiple physical drives, which can be managed separately. A partition is a logically independent section of a hard disk drive that contains a single type of filesystem.
* Volume: refer to a disk or partition that has been initialized with a file system.
* File system instance (or filesystem): refer to all the files, directories and file system data structures stored in a partition of a device.
* Metadata: is data information that provides knowledge and description about other data. For example, creation time of a file is a valuable information about a file, but it is not part of the data stored in the file.

Logical Volume (#TODO to add)

* Superblock: a small area on a volume where characteristics of the filesystem currently residing on that volume are stored. Superblock usually contains volume size, block size, layout and counts of empty blocks, with other file system specific metadata such as size and location of i-node tables, disk block map and usage information.
* I-node: is a data structure of a file system implementation, usually on Unix-like operating systems, where all information about a file except its name and its actual data are stored. The i-node also provides the connection to the physical locations on disk which have the file's data.

### Operations

This chapter introduces some basic operations involve around file system, how a file system would work based on two basic abstractions of files and directories. These operations are not necessarily corresponding to user-level operations, the former might be executed internally by the OS and do not send any notification to the users.

* Initialization:

First operation to attach file system to any fresh partition in a drive is always creating an empty file system on a given volume. The volume's characteristics like total size and user's preferences will be taken into account when creating and placing internal data structures of file system onto the fresh volume. Most of the time, this related to the size of superblock and how does files and directories will be stored later.

<fig3>

<fig4>

During the process of initializing file system, empty top-level directory must be created, also known as root directory. Every files and directories created after initialization phase are organized under this root directory in the hierarchy. In Linux world, this is usually done by executing MKFS programs from command line interface (CLI) as shown in <fig2>, whereas Windows users are more familiar with this process under the name Formatting in Formatting a disk or USB drive <fig3>.

* Mounting:

Mounting is the task of accessing the raw storage device, using file system program to read its superblock to compute necessary information for later reading and modifying memory content. Even though mounting is often a user program, the process returns a handle to Operating System for accessing the disk. In graphical OS environment, mounting is transparent and performed automatically when there is a new storage device connected to computer motherboard. One crucial job when mounting a disk is to check the consistency state of the disk to find out if it was cleanly shutdown last time, or there is a power outage while data was being written to the disk. Such inconsistency can cause corruption to the disk and all data can be lost. In case file system detects unclean shutdown state of the disk when trying to mount, extra tasks of verifying and possibly repairing any damage must take action. These tasks will be handled by another program called file system check, or FSCK for short, and it is extremely complex and takes time since the whole storage needs to be checked.

* Unmounting:

The unmount operation indicates the user no longer has intention to use a particular volume and it should not be available to read or write. The process involves flushing all unwritten data from Random Access Memory (RAM) to physical media, marking the volume to be "clean" indicating that a normal shutdown is performed and removing the accessing handle from Operating System. After unmounting, it should not be possible to access the storage device until the next mount operation.

* File and Directory Manipulation:

Users should have ability to interact with their data at file level by any actions including creating, opening, writing, reading, deleting, moving and renaming. As mentioned above, a directory is essentially a file where its data is the name list of sub-files and directories. Therefore, most file operations can also be applied to directory.

After mounting a fresh file system, there is nothing on the volume until a new file is created and some data is stored in the file to be written permanently to disk. Creating a file requires name of the file and which directory the file will reside in. If the volume is empty, new file will be created under root directory. File creation only involves allocating i-node and writing metadata of the file, the actual data, stream of bytes in the file will be added later when writing actual happens. Since having a new file under a directory means the content of the directory has been modified, file system also needs to update the parent directory's content.

When a number of files are presented on file system, they need to be opened before being capable of reading and writing. Similar to mounting file system, opening a file returns file handle to OS, instead of the volume handle when mounting, and by using this handle, the OS can request file system to read or write to a specific location in the file.

Write operation allows computer programs to store data while read allows extracting information in files. File system needs a reference to the file, which is returned from open operation, offset position in the file to begin reading and writing, memory buffer and length of the requested writing or reading data. Writing to a file can increase file size and, in such case, extra memory blocks will be allocated to append to the end of logical file memory array, which eventually update the file i-node and potentially superblock of the file system depending on its implementation. Writing to a logical address of a file will be translated to a physical address by file system, thus making it permanent. Read operation is much simpler, all file system needs to do is map logical position of the file to the corresponding physical block on disk, which comes down to retrieving data from storage device and place it to read buffer.

On the other of creating files, it is only natural to have option to delete them. File deletion process first removes the name of the file in parent directory's name list and until there are no more programs with open file handle to the file, file system can free the file's resources by returning occupied blocks to the free block pool and the file i-node to the free i-node list. It is not necessary to set all the memory bits associated with the metadata and data of the deleted file to be zeros. Marking a delete flag in the file metadata and free these blocks in used/free block pool is sufficient for the OS to know previous memory areas are free to be overwritten.

Rename is often regarded as the most complex operation that a file system has to support. There are many checks and validations must pass before actual renaming takes place. These checks include whether the source and destination names are in different directories, then new name can be the same as the old one, if names refer to directories, new name cannot be sub-directory of the old name, etc. Only after all validations are satisfied, file system will delete old name entry, create new name entry in new destination directory then update corresponding parent directories.

### Existing File Systems

File systems are usually packed within OS and each OS has their own implementation of file systems. Depending on the underlying physical storage medium, purpose when storing data and host OS, file system type will be chosen for individual media. For instance, CDs use ISO 9660 file system <ref9>, hard disks and managed flash drive like SSD depend on what is available from the host OS, Microsoft Windows utilizes NTFS <ref10> while MacOS takes advantage of Apple's APFS with improved GUI experience <ref11> <ref7,37> and Linux has ext4 developed by the Open-source Community <ref12>.

<fig5>

In summary, file system is a program that enable Operating System to store, retrieve and manipulate data on persistent storage device, data here is usually stored as content of files and organized into directories. Each file system targets a specific type of storage on a specific Operating System, though they all must support some basic operations to comply with existing conventional and modern designs.

## Block-based File System

During the development of modern computers, tech researchers and programmers have experimented with many file system designs, investigated both simple and complex solutions, however, block-based type file system, which is also known as traditional design, stood out both in terms of simplicity in implementation and effectiveness in performance. Block-based file system is straight-forward when coming to explanation of the concept, storage is divided into blocks and file need to be stored in blocks boundary <ref13>. Any modern file systems based on this central design can trace their inspirations back to Berkley Software Distribution Fast File System (BSD FFS), who set the robustness and speed standards for Unix file systems in nearly a decade. Traditional design usually consists of a superblock, a block bitmap, a i-node bitmap and a data mapping scheme in form of direct or in-direct blocks. This design can be found in many popular file systems in computer history such as Be File System (BFS), Second Extended File System (ext2), and File Allocation Table (FAT) File System. <ref1 p-33-35>

### Internal Components

The following section introduces underlying basis of block-based file system. Most of these file systems share the same notion for some internal components which they comprise, for example: superblock, bitmap, i-node, and data stream.

Essentially, there is a known place on a storage device to store the most important block to the file system called superblock. Suitable choices for this place are often the first or the last block of the partition but first block is usually chosen for the ease of seeking (no need to find the size of the partition to calculate the last block number). In superblock, it contains information about block size, total number of blocks, number of used blocks, dirty bit flag, and address of i-node of the root directory. Without this reference from superblock to the root of the hierarchy of all files and directories, the file system would not able to connect and find any files on the volume.

Following blocks after the superblock can be occupied for bitmap scheme, which is an approach to manage free space of a disk. The bitmap scheme represent each disk block as 1 bit, thus binary value of 0 or 1 in a bit can indicate vacancy status of a block whether it contains invalid data that the file system can freely write or it is not empty and should not be overwritten. The number of blocks used for bitmapping is totally based on the size of the partition and file system block size, as calculated in Equation 1. Each byte consists of 8 bits, therefore the bitmap for 8GB disk with 1K blocks would requires 1MB of space, or 1024 blocks in this case.

<equa1>

Beside metainformation about the file such as the size of a file, access permission information, its creation and modification times, i-node data structure needs to keep track of which locations on disk are belongs to this i-node data stream. This basic structure is the fundamental building block of how data is stored in a file on a file system.

Traditional approach for linking on disk addresses to logical file offsets is storing a list of blocks directly inside i-node, which is called direct blocks. Each entry in this list is a physical block address of the storage device, and since the size of an i-node structure is limited, it limits the amount of data the file can contain. Generally, about 4 to 16 addresses can be stored directly in i-node, which means maximum size of a file can be is 16KB with 1KB file system block size.

<fig7>

To address the space constraint of direct block, same concept can still be applied with extra address block in between, which called indirect block. Rather than having direct reference to a physical address that has the file data, i-node can carry block address of this indirectly block. While data block contains user data, indirect block has pointers to other data blocks that do have user data in them, which make up the whole stream bytes of file data after concatenating. Therefore, one disk block address can map a much larger number of data blocks, instead of mapping one by one with direct block address.

<fig8>

Indirect block helps increasing the maximum data size of a file an i-node can keep track, however, it is not enough to locate data blocks for a file which is much more than a few hundred kilobytes in size. To overcome this issue to allow an even bigger file, indirect block technique can be applied a second time, making double-indirect blocks. The same concept and basic idea still hold true for double-indirect block as indirect block. Each double-indirect block address that i-node contains points to a block on disk whose content is more pointers to indirect blocks and respectively refers to exponentially amount of actual data blocks constructing a file.

### Advantages

robust, straight-forward easy to implement

Traditional design of file systems is straight-forward, there are not many abstraction layers between how users see files and directories on computer screen and how file system hides its implementation details to store those data on disk. Basically, data in a file is broken down into many chunks of blocks and file i-node is responsible for keeping track those blocks locations. On top of that, file system superblock will always have connection to the address where each i-node resides and return that information whenever there is a request to read/write that file. Subsequently, when accessing a file, in addition to disk operations of accessing directly the blocks having actual file data, there will be few more reads to superblock and other blocks to search for the i-node and eventually leading to the data blocks. These additional reads are called file system overhead, and because of the simplicity in the design, traditional file systems performance suffer little from it and tend to be robust since less complexity means less bugs and corruptions. When putting more optimization features such as bigger such as block size and caching, performance can even be pushed further.

### Disadvantages

Writing any new data to the disk might require updating file system housekeeping data structure, which are superblock and bitmap, and having these areas to get frequently updated is a big disadvantage in this design. It really depends on the underlying storage device, continually changing the content in a part of a hard disk drive might not be a problem, but doing the same thing in modern flash drive is not at all recommended due to the drive's characteristics and operations involving in writing data to a memory block which will be discussed in Chapter 2.3 about Flash Memory. One key disadvantage of flash memory is that it can only endure a relatively small number of write cycles in a specific block. Therefore, block-based file system is not a suitable choice to be placed on top of flash storage device.

## Flash Memory

#TODO many plagiarism

Flash memory is an electronic non-volatile computer memory storage medium that can be electrically erased and reprogrammed. Flash storage device is the successor of hard disk drive, which use mechanical movement to read and write memory <todo>.

Non-volatile storage technology that does not require power to retain data

Was invented by Toshiba in 1980 based on EEPROM technology

Toshiba introduced NAND Flash for the first time in late 80s

Individual flash memory cell consisting of a FET transistor and floating gate

Floating gate is used to store cell's value

No electrons on the floating gate -> cell is in the erased ("1") state and has a low "turn on" threshold voltage (Vth)

Electrons on the floating gate increases Vth voltage -> Cell is in programmed ("0") state

Write operation:

A voltage aplied to the control gate causes a tunnel current to flow through the oxide layer, thereby injecting electrons into the floating gate

Erase operation:

A voltage applied to the silicon substrate releases the electrons accumulated at the floating gate

Flash gets its name from the requirement to do all bit bulk flash erase of the whole sector, block to bit 1

Flash memory is a modern type of computer memory constructed from semiconductor components and used primarily in persistent storage mediums. This memory type falls into category of non-volatile technology meaning that power is not required to retain data in the memory making it a perfect match to use as secondary storage, or long-term persistent storage.

<fig8> <ref15>

<fig9> <ref15>

Flash memory was invented by Fujio Masuoka while he is working at Toshiba in 1980 <ref14> and made commercially first introduction to the market in late 1980s <ref14>. The key component that composes individual flash cell is floating-gate metal-oxide-semiconductor field-effect transistor (MOSFET), also known as a floating-gate transistor. Differentiate from a normal MOSFET shown in <fig8>, a floating-gate MOSFET has an extra gate called floating gate added between the control gate and the body of the transistor as indicated in <fig9>. The floating gate is separated from the control gate and the body by the oxide insulating layers preventing any electrons on floating gate to escape easily. Having this electrical isolated element allows charges on floating gate to stay for long periods of time, and flash memory utilizes this characteristic to store cell’s value on floating gate. Theoretically, if there is no electron on the floating gate, the cell is in erased state and the bit value is known to be 1, whereas having electrons on floating gate presents bit value of 0. <ref15>

NOR flash 1984

NAND flash 1987

### Operating Principle

each cell > floating trnasistor > 2 charge states > 1 binary (1 when no charge, 0 when) > SLC > more than 2 charge state > more than 1 bit encoded per floating gate > MLC more than 1 bit per cell > smaller tolerance (increase Vth voltage) > reading > negative charge on floating gate screen off some positiv charge on control gate > need more charge to reach threshold > current vs gate to source voltage plot > aplly intermidiate voltage in between 2 threshold voltage and measure the current > same thing apply for mlc flash + plot > writing (moving charges to and from the floating gate) > program (inject electrons into the floating gate '1') > erase (release electron accumulated at the floating gate '0') > 2 methods (not go into details) > limitation

Flash memory stores information in an array of memory cells made from floating-gate transistors. each cell stores only one bit of information in single-lvel cell (SLC) devices, while multi-level cell (MLC) devices, TCL, QLC, PLC can store more than 1 bit per cell.

Flash memory is based on floating gate transistors, which is a variant of MOSFET with one small change. There is an extra gate that is added between the control gate and the body, this extra gate is known as the floating gate is electrically isolated. It had no electrical contacts which means that any charges put on the floating gate will stay there for a long time (for years in fact, and that's what allow flash storage to store data without the need of power source.

A flash memory device has all the data encoded as a bunch of charges on these floating gates.

How read and write operations in flash storage device are performed will be discussed in this section with principles down to flash memory cell level. As previously introduced, each memory cell consists of a floating-gate transistor which can have two charge states on the floating gate, either no charge citing bit value of 1 and negative charge implying bit value of 0, and this type of two states cell is called single-level cell (SLC) and capable of storing one bit of data. However, to achieve larger bit density in a single flash integrated circuit (IC), multi-level cell (MLC) technology was developed to allow more than one bit to be encoded per floating gate, resulting in increasing maximum capacity of flash memory significantly. The theory behind MLC is based on the fact that the number of electrons which can be charged into the floating gate is a variable. Therefore, instead of having only charge and no charge states, floating gate can analogously have as many charging states as it wants depending on the amount of electrons presented on floating gate. Even though technology always seeks the greatest number of possible states, it comes with a downside when evaluating state of floating gates. More charge states express smaller tolerance in distinguishing them leading to higher chance of errors both in determining the amount of charges and injecting correct number of electrons. Considering the size of a typical flash cell is only ??nm <ref16>, measuring any aspects of the cell requires extreme precision. Until today, most common types of MLC flash devices have 2 to 4 bits per cell, which requires 4 to 16 states of the floating gate to be distinguished <ref17>.

<fig10> more bits means smaller reliability margin

Reading the value of a cell is basically detecting which charge state the floating gate is having, and from that charge state, bit value can be translated. Based on the principle operations of MOSFET, if there is a threshold voltage applied to the gate, current will flow from source to drain <ref18>; applying some voltage to the control gate of the floating-gate MOSFET, the existent of current from source to drain can disclose the charge state of the floating gate. Electrons on floating gate act as a negative mask layer which screen off some positive charges on the control gate, subsequently having effect on controlling the current. The strength of this influence depends on the amount of negative charges on the floating gate. Therefore, the voltage required for the control gate when the bit value is 0 is higher than when the bit value is 1 (no electrons on the floating gate) in order to have current flows in the drain terminal. In <fig11>, the threshold voltage for erased state is called VT1 and threshold voltage for programmed state is called VT0; and according to earlier statement, VT0 > VT1. <ref18>

<fig11>

<fig11> demonstrates the relationship between the current flowing from source to drain versus gate to source voltage in an IV (Current vs Voltage) graph with two “turning on” curves starting at VT1 and VT0, corresponding to two threshold voltages of different charge states. An intermediate voltage in the middle of VT1 and VT0 indicating by the blue line on <fig11> can be applied to measure the current. When identifying the charge state on floating gate to determine the bit value of a flash cell, no reading on current value means the transistor is on “0” curve correlating to bit value 0, while positive current reading means the other curve and bit value is 1. <ref18>

<fig12>

The approach for MLC flash memory is not too different, instead of two curves account for two threshold voltage levels, there are four curves, one for each charge state, present in the IV graph <fig12> if each cell encodes 2 bits of data. On this condition, there are three intermediate voltages and by reading the current value when applying each one voltage difference from gate to source, bit values can be identified.

<fig13> <ref20>

<fig14> <ref21>

Beside reading, the other fundamental operation on a flash cell is writing, which is the process of alternating number of elections on floating gate to reach different charge states. Despite no electrical contact to the floating gate <ref?>, few methods of sending charges through the oxide layer has been discovered. First method is Quantum Tunneling and the second one is Hot Electron Injection. This document is not addressing these subjects in details; however, in short, Quantum Tunneling is based on quantum physics to shrink the oxide gap between control gate and floating gate to push electrons off from control gate to floating gate and vice versa <ref20>, whereas, Hot Electron Injection increases kinetic energy of electrons from body substrate to let them hop over the insulating layer onto floating gate <ref21>.

### Hardware Limitation

Wearing

Equal distribution, load balance

Both writing methods (quantum tuneling and hot injection) involve high voltages and high electric fields and this limits the number of times you can write to the floating gate transistor. What happen is that the electrons gain a lot of energy and dissipate that energy by colliding with the oxide layer lattice and this damage builds up over time. Oxide insulator wears out over time causing electron leakage that alters floating gate charge. Defined Vth limits are no more accurate and value detected wrongly. Once you've done enough writes, the damage is great enought taht the device becomes unusable.

SLC 100,000 writes, MLC 1000 - 10,000. That's small is because of the lower tolerance is involded for mlc so it takes less wear to make MLC unsuable.

Because of the limited number of writes flash flash memory needs to have some sort of wear leveling.

If the wear leveling wasn't there you might end up writing to the same area to the same block of memory over and over again and make that block of memory unusable very quickly, so we're leveling everns out the load so you don't get one region of memory wearing out quickly.

Don’t get wearing out much quicker than the rest of the flash

Reduce capacity of the flash

Flash technology is gaining its popularity and slowly becoming the successor to replace hard disk drives for secondary storage due to its faster read and write speed since there is no mechanical part movement associated like in the latter <ref?>, though one major limitation it is having is flash memory cannot stand the test of time. This issue of the technology is called memory wear, since flash memory has a finite number of program – erase (P/E) cycles before it becomes unreliable for reading. The reason behind memory wear connects directly with the current ways how the memory cell is written. Both writing methods (Quantum Tunneling and Hot Electron Injection) require high voltages and high electric fields to be applied on the transistor, which causes the electrons to gain huge energy then dissipate that energy by colliding with the oxide layer lattice <ref?>. This damage builds up overtime and oxide insulator wears out eventually, causing electrons leakage on the floating gate, thus charges get altered unintendedly and bit value is encoded wrongly.

<fig13>

On MLC flash memory, wear problem has even greater effect due to the small tolerance of the reliability margin. Illustrated in <fig13>, threshold voltage region of each charge state is much smaller than in SLC memory and potentially crosses over others when worn out, represented by dotted grey line, hence tiny error in number of electrons can cause incorrect reading when applying the same factory defined intermediate VTHR voltages for measuring. Apparently, due to the smaller tolerance involved for MLC, it takes less wear to make MLC flash unusable. Typical maximum number of writes before wearing happens for SLC is 100,000 P/E cycles while this number for MLC only ranges from 1,000 to 10,000 cycles <ref22>.

<fig14> <ref23>

The consequences from wearing effect only rise when some blocks in the whole flash memory has significant higher erase count than the rest of the blocks because their data were constantly focused for rewriting. The heat map in <fig14> show an example for such situation where about 20% random locations in the flash is reaching the maximum P/E cycles and other flash blocks still has considerably low erase cycles. In such condition, once those higher rewritten rated blocks wear out, the whole filesystem residing on this flash memory is acknowledged to be corrupted and the flash is not meant for continued use, even though 80% of the flash is still in good shape. Therefore, it is not a good use case of flash memory to get frequently updated and rewritten partially, this will decrease the flash’s life span tremendously. In order to extend practical used time of flash memory, data written to the storage should be distributed equally to all blocks, thus having erase counts across the flash to be closed to each other. <fig15> portrays what a healthy flash memory array should look like when all blocks have approximately similar P/E cycles.

<fig15> <ref23>

To guarantee written data is distributed equally across the whole flash, at least one type of wear levelling should be present in the whole system. Wear levelling solution can be accomplished in form of hardware as an extra IC in the same package of the flash or executed as software directly from the host computer. In summary, if no wear levelling was integrated in a system that use flash storages to even out the load, applications might end up writing to the same block of the memory repeatedly, thus making that block of memory unusable quickly.

### Managed Flash vs. Unmanaged Flash

Modern flash storages can be divided into two types, managed flash and unmanaged flash, based on the existence of a component called controller in the same storage package.

Popular memory devices such as SSD, SD, eMMC, UFS found in PCs, laptops, mobile phones nowadays are all managed flashes. There is a controller on the device that do many memory management applications specifically for the current type of flash such as error correction, wear leveling, bad block management, etc. By having a dedicated controller for the flash in the same package, all management operations are handled by the memory internally, lifting the burden from host computer to take extra care for any of these services. Moreover, integrated controller has better understanding of the underlying structure and characteristics of the flash than any external controller can possibly support, proving that internal controller is likely to have better optimizations at doing any mentioned operations.

Unmanaged flashes are also known as raw NAND/NOR chips, which essentially consists of only memory array ICs without any kind of dedicated memory management component. This type of storage tends to have smaller footprint and lower cost comparing to its managed counterpart, thus suitable for embedded applications where they are used in an often limited resource systems. Since there is no onboard memory management, all responsibilities to provide these instructions rely on host side. Therefore, when implementing file system targeting to use for such devices, wear leveling needs to be taken into consideration somehow that data will be written evenly to all blocks of the flash. If the design of the file system is comprehended with traditional approach described in Chapter 2.2, first few blocks of the flash where superblock and bitmap are resided will have higher P/E cycles than any other blocks.

Below graph shows an example when using FAT file system (which uses some beginning blocks for storing file system metadata, i-node table in this case) directly on a raw flash without wear leveling, which should always be avoided.

<fig16>

To sum up, traditional file systems are inadvisable choices to manage files on top of raw flashes because of an obvious reason, they are wear-levelling intolerant. Therefore, new file system designs need to be invented to conform with the natural characteristics of flash memory.

# Design and Implementation

This chapter contains an overview of the design for SPI NOR File System (SPINFS), a file system the author has developed and named according to a specific targeted type of flash memory: raw NOR flash with SPI communication. As described in the Introduction, the goal is to implement a file system that could overcome wear behavior of flash memory to achieve longest possible lifetime and assure certain level of data integrity. First, there is a brief introduction to the structure of the file system and its internal components. Afterwards, operations of SPINFS will be introduced with their principle based on underlying data structure and concept. Finally, details for a special operation which is fundamental for the design of SPINFS called Garbage Collection will be introduced

## Design Overview

Conceptually, SPINFS is a log-structured file system and takes a completely diverse approach to popular file systems on hard disk drive like BSD FFS and FAT. Differentiating from traditional block-based file system that storage location is bound to a piece of data, log-structured file system makes use of entire storage for a circular log which is appended with every change made to the filesystem.

<fig17>

<fig18>

Unlike their block based counterpart, which has i-node blocks representing a file or directory scatter around the whole disk with address pointer pointing to the blocks that have the actual data, which also spread out; the circular structure consists of multiple nodes, each representing a file or directory and the data inside it, following each other. To simplify, traditional file system has files laid out in a predetermined way with space between them as shown in <fig17>, while <fig18> demonstrate principle operation of log-structured one, thus files are written one by one in ascending order to the end of storage media.

## Node Data Structure

In SPINFS, the basic building block in the log is a structure known as **struct spinfs\_raw\_inode** which also is the only type of node in this design of file system. Because file and directory are treated equally in SPINFS, only one node structure can be considered to use for both, which minimize the complexity of its architecture. Directory is theoretically a file whose data is a name list consisting of files and sub-directories are under this parent directory. Moreover, both share some similarities in metadata footprints, for example, name, i-node number, creation, modification time, size, owner and access flags are some matching statistics of file and directory. Therefore, detailed members of SPINFS node structure are displayed in Listing 1.

From these matching statistics of file and directory, spinfs is considered to have only one raw\_node data structure that can be used for both of them, which minimize the complexity of its design.

The structure starts with a common header containing the i-node number of that node and all metainformation for that i-node, and may also carry a variable amount of data.

<list1>

SPINFS node is very similar to conventional i-node that it contains the metainformation about entities that live in the file system with only one exception; instead of storing addresses of that entity's data blocks which live somewhere else, data part is included directly inside the node structure, right after metadata segment.

Looking at the raw i-node structure, it is clear to see some basic file information that SPINFS supports. First of all, maximum file name length is 32 one-byte ASCII characters, which is said to be the minimum requirement for any interactive system <p18. Ref7>. The next field, **inode\_num**, is self-explain, it is the i-node number of file or directory current node is referring to. In SPINFS, 32-bit unsigned integer value is used for storing i-node number, allowing for 4 milliard files to be existed on the filesystem. However, considering that raw flashes are usually appeared in embedded systems, this maximum number of files are unlikely to be reached due to the simplicity and often single, straight forward application these kinds of systems provide <ref24>. One thing to noted is i-node numbers are never reused in SPINFS. In case when files are deleted, new files will always have the next highest available i-node number rather than having an obsoleted number from deleted files.

The **uid** and **gid** fields record ownership information about a file. SPINFS is designed to be run on Linux host machines, thus it follows convention that was specified in Portable Operating System Interface (POSIX) compliant that any files must have corresponding user id and group id which this file belongs to <ref25>. Combining with **mode** field, the file system can provide file access permission check. By following POSIX specification, file permission model in SPINFS consists of user, group and "other" classes and three distinct operations that these classes can do to a file system entity: read, write and execute. The checking is done by comparing current logged in user with **uid** and **gid** fields to determine which class file access permission will be checked, then **mode** field will tell if the user can continue doing what he/she intends to do with the file, either read, write to it or execute an executable file. In addition to file access permission, SPINFS also stores information about whether this node is a regular file or a directory along in 32-bit **mode** value. (INTEGRITY libguide p.96 file system terms)

Moving on to the next **flags** field, it is a record of various bits of i-node states, and at the time of development there are only two other states beside normal state of an i-node in SPINFS. With two extra obsolete and deleted states, this field will only need to provide 2 bits in total of 32 to indicate the status of i-node.

SPINFS maintains some useful timely aspect of files known as creation time (**ctime**) and last modified time (**mtime**) of a file, and with these information users can easily query for files with a timestamp in mind. Unlike others Unix file systems, the author’s approach does not attempt to support last accessed time (**atime**) simply because this information is too expensive to maintain over the small amount of benefit that it provides. Every access to a file will need to update its node structure, and in a log-structured file system, this means the whole new node with all file content will be written to the flash even when only a small part of the file is read, which eventually wears the flash extremely fast.

The **parent\_inode** field is effectively the i-node number of the parent directory of current file or directory. With parent i-node stored directly inside a node structure, traversing backward the filesystem tree can be made efficiently to reconstructing a full path name of the file, while without it, the only way to know the full path name of an i-node is having that information in memory all the time while the file is opened.

The **version** field is a crucial piece of information for any nodes because it is where SPINFS maintains historical ordering for them. As the spirit of SPINFS is a log-structured file system, every update to a file is actually writing a new node of the file to storage rather than modifying old data blocks, the file system needs a way to identify which is the most recent **version** of a file among many nodes referring to the same file. It is exactly what this field in raw i-node structure is responsible for, each new node is appended with a **version** higher than all previous nodes belonging to the same i-node number. Similar to **inode\_num**, version is an 32-bit unsigned integer, hence there is a ceiling limit for a number of nodes can be updated for an i-node during the whole flash chip's lifetime, however, this amount of updates is reasonable so this limitation is deemed to be acceptable.

Last two members in the data structure specify data segment of the node. The **data\_size** field indicates how many bytes is the size of the node, which essentially is the size of a file in case this node is a regular file node. Finally, the last field is a flexible array member, **data[]**, where the actual data stream of a file is located. Since files' size varies, length of **data[]** array is determined by **data\_size** field, thus the starting address of the data is always presented in the node structure while the last address can be easily calculated for the file system to seek the next consecutive i-node.

After some considerations and updates to the design of SPINFS, the author decided to implement checksum for each node to improve integrity of the file system. In C programming, flexible array member has to be the last data member of a struct <ref 26> so a new checksum member value cannot be added to the end of the current data structure, thus leaving the author of SPINFS no choice other than having it in data[] array right after the actual data of the node. Consequently, this increases the size of each node by the size of the checksum, 160 bits for SHA1 sum in this case <ref27>, at the end; hence making **data\_size** no longer reflects the size of a file if raw i-node is a regular file node, but the actual size plus 20 bytes.

By always writing to the end of the flash in circular motion, all blocks in the flash will be written at the same rate and have even number of erase counts

## Operations

mkfs.spinfs

In order to use any file system with a storage device the very first operation is initializing the medium with a format that file system can understand. According to spinfs, this step is essentially erasing the whole raw flash device and write the first node for root directory, which is also the most important entity in any file system. Writing root directory can be as simple as transfering all bytes consisting in a raw\_node structure with "/" name, i-node number of 1, parent i-node of 0, since root directory has no parent, and data part has 0 byte size to address 0x0 of the flash.

create

The next usual operation with file system is populating the storage by creating files or sub-directories under root directory. New file created in the filesystem means a new node structure referring to the file with corresponding i-node will be written to the flash, continue after the last occupied address of the last node, in the case with a freshly formatted filesystem, new node is added after root directory node. Continue creating new files and sub-directories will keep appending new nodes for those entities to the flash toward the ending address. At the point when there is not enough space to allocate new node, an operation called garbage collection will be triggered trying to search for free space, and if it fails to do so, a "Not enough space left" error will be returned. Garbage collection will be discussed more in details later in this section.

update

Due to the design of log-structured file systems, every update to content or metadata of a file will result in writing a whole new node for that file with modified data to next available space in the raw flash. The new node will have higher version value than the old ones, and these old nodes are said to be obsoleted, where the content they contain has been outdated by a later node. Space taken by obsoleted nodes is referred to as "dirty space" and will be reclaimed later by garbage collection operation.

delete

File deletion is also considered an update to the file. Spinfs simply marks the file as deleted by setting deleted bit in its flag and write a new version of the file's node with zero data to the flash. After the file is marked as deleted, any operations with associated i-node number should not be possible and return error.

note about file/dir creation

When creating or deleting any files and directories, content of the parent directory is subjected to get modified, hence an update to the parent's node will be performed and another new node will be written.

read

need to find the latest version of the node

For the reason that there will be many nodes associated with a single i-node number in file system structure, when reading content of an i-node, spinfs should ignore all obsoleted nodes and only return the latest information about that i-node. Therefore, spinfs always keeps an i-node table in memory and updates this table with corresponding changes. The i-node table has columns for i-node number, latest version of raw\_node for that i-node and physical starting address of that raw\_node in storage device. At mount time, spinfs scans through the whole flash to populate this table. Upon looking at a new i-node number, a new entry will be appended to the table, while upon a newer version of an entry is found, that entry will be updated with the corresponding data.

## Garbage Collection

head and tail store in Security Register, int32, so can store # slots,

At the point when the system is out of space due to continuous writes of new data, it needs to start reclaiming the dirty space which is the result of obsoleted and deleted nodes. To assist this reclaiming operation, the file system always keeps track of physical offsets in flash media of the oldest node and the next erased address for a new node, these address offsets are called head and tail respectively. In a filesystem that this cleaning process has never been triggered, the head will stay at the very beginning of the flash, whereas the tail is reaching the last available address.

(Should the following paragraph be in Testing Hardware section)

During practical work of this project, the test flash chip model has feature called Security Registers, which are small separate memory regions aside from the main memory region consisting of only 256 Bytes, and two of these registers are chosen solely for the purpose of storing the head and tail pointers’ values. In spinfs, head and tail are 32-bit unsigned integer, thus each Security Register can store 64 values for them and will be updated according to each write to main memory region. This means any new node written to the end of the flash will also write a new value for both head and tail in Security Registers 1 and 2. When all 64 slots has been used, spinfs simply erased these two registers and start writing new head and tail values from the beginning. The search for correct, or most recent, head and tail values in Security Registers is straightforward since the registers can be read backward from the end and the first non-erased value, not 0xFFFFFFFF, is the wanted one.

Each time garbage collection code is executed, the objective is to erase the first flash block pointed by head. The operation will begin by iterating head toward tail in natural direction over each node and examine whether the pointing node is obsolete or valid. In case the node is obsoleted by a later version written with the same i-node number or it is marked with a deleted flag with zero sized data part, it will be skipped, and the head moves on to the next node.

On the other hand, when the node is still valid, meaning that it has the latest version of a still-in-use file, the whole node will be copied to the tail of the log by writing an exact same node just with bigger version number, hence rendering the current node to obsolete and the head continues iterating.

Once the head has been progressed to the next erased block boundary of the flash, all nodes in the previous block should not be relevant to the filesystem anymore and it is safe to erase that block without losing any data. As a result, the amount of free space in the flash, indicating by number of bytes counting from the tail address to the head address, increases by an erased block size. In instance of insufficient dirty space to have size of one erased block when added together, garbage collection process will return error.

Don’t have time to do. Put it in future development

# Testing result

## Hardware in use

This project was developed on a Raspberry Pi and a NOR flash chip soldered to its breakout circuit board which then is connected to GPIO pins corresponding for SPI communication channel 0 on the Pi. The specific model of the Pi is Raspberry Pi 2B #FIXME, and the flash is NOR Flash IC S25FL164K from Spansion #FIXME. The Pi is acting as the host running Raspbian which is a distribution of Linux Operating Systems targeted to Raspberry Pi’s hardware. From the breakout board of the flash chip, jumper wires were used to connect directly to GPIO pins on the Pi as shown in <pic> and described in <table>. Because spinfs was written with C programming language, there is a need to interface SPI communication to the code, and the author has chosen to use SPI APIs from WiringPi library. For example, to send over a SPI packet, programmer can call wiringPiSPIDataRW(SPI\_CHANNEL, buf, sizeof(buf)) function. This library can easily be installed from apt-get package manager in Raspbian and should be linked with when compiling C source files with “-lwiringpi gcc” option.

The S25FL164K flash model is a 64 Megabits (8 Megabytes) variant in its flash chip family, thus having address counting from 0x000000 to 0x7FFFFF. All flashes in this family share some similar characteristics, such as 256 bytes programming page size and smallest erased size is 4 Kilobytes. It is a NOR flash meaning any byte in the flash are allowed to get accessed directly for reading and writing; comparing to NAND flash, this has to be performed on the whole page. As being said, a read command can read one byte at any specific address or data in following addresses will also be returned as long as clock line in SPI protocol is still running. However, program to the flash accept only 256 bytes at most <ref. 2>; and if this maximum value is desired to be written, the address used in the program command should be 0xXXXX00, otherwise the address will be wrap around at page size and old data will be overwritten.

<listings about Defined flash information>

## Functionalities

Ls

cp

# Discussion

## What design goal has been achieved

Wear leveling

Data integrity with checksum, tail update after writing new node, so in case of power failure during write, nodes before tail are always valid (code challenge to verify this and clean up bad write)

## Limitation of current design

write to flash every update

Not power cut tolerance

## Further development

## What I achieve with this project

# Conclusion

References

Details of the references are given here. Use the referencing system required in your degree programme or as agreed with your supervisor.

Layout of this page in the author-date /Harvard) referencing system:

Details of the reference Details of the reference Details of the reference Details of the reference Details of the reference.

Details of the reference Details of the reference Details of the reference Details of the reference Details of the reference.

Layout of this page in the number (Vancouver) referencing system:

1. Details of the reference Details of the reference Details of the reference Details of the reference Details of the reference.
2. Details of the reference Details of the reference Details of the reference Details of the reference Details of the reference.

Title of the Appendix

Content of the appendix is placed here.

Title of the Appendix

Content of the appendix is placed here.